VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Srovnání výkonu implementací hash tabulek při paralelním přístupu

Comparison of Hash Table Implementations Performance using a Parallel Access

Dominik Meca

Bakalářská práce

Vedoucí práce: doc. Ing. Radim Bača, Ph.D.

Ostrava, 2021

Abstrakt

Cílem bakalářské práce bylo srovnat výkon několika implementací hash tabulek při práci s více vlákny. Úvodní část práce se věnuje hash tabulkám a porovnává jejich vlastnosti s dalšími, často používanými, datovými strukturami. Zbytek práce se věnuje specifickým implementacím jednotlivých tabulek a ukazuje výsledky třech různých testů.

Klíčová slova

hash tabulka; práce s více vlákny; porovnání výkonu; Libcuckoo; Junction; Intel oneAPI Thread Building Blocks; C++

Abstract

The goal of this bachelor thesis was to compare the performance of several hash table implementations under a multithreaded workload. The introductory part of this work describes common hash table implementations and compares them with other popular data structures. The rest of this thesis describes the algorithms used in several different hash table implementations and shows the results of three separate benchmarks.

Keywords

hash table; multithreading; benchmark; Libcuckoo; Junction; Intel oneAPI Thread Building Blocks; C++

Poděkování

Rád bych poděkoval panu doc. Ing. Radimu Bačovi, Ph.D., za jeho ochotný a vstřícný přístup při vypracovávání bakalářské práce.

Obsah

Seznam použitých zkratek a symbolů

RAM	–	Random Access Memory
DDR	–	Double Data Rate
SDRAM	–	Synchronous Dynamic Random Access Memory
WG21	–	Working Group 21
oneTBB	–	Intel oneAPI Thread Building Blocks
QSBR	–	Quescient state-based memory reclamation

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Trend explozivního růstu výkonu procesorů a rapidně se zmenšujících komponentů a součástek, který nás doprovází už přes čtyřicet let, je na ústupu. Frekvence veřejně prodávaných procesorů se za posledních deset let zastavila u hranice 4GHz, kterou procesory překročí jen na malý okamžik s technologiemi jako je Intel Turbo Boost nebo AMD Turbo Core. Nadějím na obnovení dřívějšího růstu výkonu nepomáhá ani fakt, že velikost dnešních tranzistorů (v řádu desítek, až jednotek nanometrů) se rapidně blíží k bodu, kde kvantové tunelování znemožní jejich fungování. Nejen z těchto důvodů se dnes výrobci procesorů zaměřují i na další aspekty, jako například snížení energetické spotřeby a produkovaného tepla. Jedním z nejdůležitějších faktorů výkonu dnešních procesorů je i počet paralelních jáder.

Větší počet jáder a programování s více vlákny ale není magickým řešením všech problémů a přináší své výhody a nevýhody. Paralelní algoritmy často bývají méně intuitivní a některé problémy jsou složité, nebo zcela nemožné, zrychlit či vyřešit. Tato situace je zhoršena tím, že velká část návrhářů programovacích jazyků volí nejjednodušší řešení a do svých jazyků přidávají jen základní synchronizační primitiva a datové struktury, ke kterým nelze bezpečně přistupovat z více vláken. Ukázkou této situace je vysoká popularita jazyků Python, kde Global Interpreter Lock brání souběžnému běhu více vláken, a JavaScript, ve kterém Web Workers fungují jako samostatné procesy bez sdílené paměti.

Vývoj implementací datových struktur, které bezpečně podporují souběžný přístup, pak spadá na samotné uživatele programovacích jazyků. Tato řešení ale často bývají vytvořené přímo na míru, mají mizivou dokumentaci, nesmyslná rozhraní, složitou instalaci, špatný výkon nebo libovolnou kombinaci z těchto a jiných nedostatků.

Druhá kapitola se zaměřuje na hash tabulky a porovnání vlastností s jinými, často používanými, datovými strukturami. Třetí kapitola popisuje a porovnává implementace testovaných hash tabulek. Další kapitoly se věnují testům výkonu tabulek.

Kapitola 2

Datové struktury

Datové struktury jsou jedním ze základních stavebních bloků každého programu. Vývojáři poskytují efektivní a prediktabilní způsob manipulace s větším množstvím dat. Různé datové struktury se podle jejich návrhu liší nejen rychlostí, ale také operacemi, které lze na datovou strukturu použít.

2.1 Praktický vliv architektury počítačů na výkon datových struktur

Ačkoliv jsou různé datové struktury a algoritmy převážně vytvářeny nezávisle na konkrétním prostředí nebo hardwaru, právě tyto praktické detaily mají nesmírný dopad na jejich výkon a použitelnost. V praxi jsou často algoritmy, které efektivně využívají vlastnosti architektury počítačů výkonnější (nebo paměťově úspornější) než algoritmy, které jsou teoreticky lépe navržené, ale tyto detaily ignorují.

Jedním z hlavních faktorů ovlivňující výkon dnešních počítačů je obrovský rozdíl v růstu rychlosti procesorů a pamětí. Zatímco rychlost procesorů se dlouhou dobu efektivně zdvojnásobovala každé dva až tři roky, rychlost pamětí se zdvojnásobuje zhruba každých pět až šest let. V procesorech se sice nachází extrémně rychlé uložiště ve formě registrů, ale jejich vysoká cena a složitá implementace neumožňuje jejich použití k vyřešení tohoto problému. Relativně pomalá rychlost pamětí RAM je o to problematičtější v dnešní době, kdy i malé aplikace pracují s gigabajty dat. Není tedy divu, že výrobci procesorů již delší dobu do svých produktů zabudovávají mezipaměť.

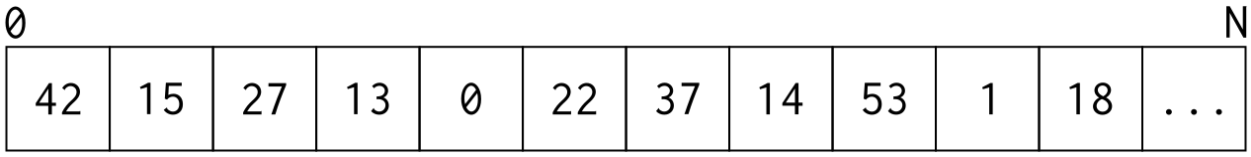
Tato mezipaměť, někdy také cache, uchovává oblast paměti, ke které procesor často přistupuje. Na rozdíl od paměti RAM je zabudována přímo do procesoru a obchází potřebu použití poměrně pomalé sběrnice. Dnešní procesory často mezipaměť rozdělují na několik úrovní. L1, nejnižší z nich, bývá rozdělena na mezipaměť pro procesorové instrukce a data. Obecně platí, že s vyšší úrovní (L2, L3, …) se mezipaměť zvětšuje a zpomaluje. Bohužel ani největší z mezipamětí nedokáže zrcadlit gigabajty paměti RAM dnešních počítačů. Pokud se procesor pokusí přečíst hodnotu z adresy, která se v mezipaměti nenachází, je nucen hodnotu načíst z podstatně pomalejší paměti RAM.

2.2 Asociativní pole

Asociativní pole je abstraktní datová struktura, kde každá hodnota je uložena podle přiřazeného klíče, který se v poli může vyskytovat pouze jednou.

2.3 Pole

Pole je jednou z nejjednodušších, nejdůležitějších a nejvíce používaných datových struktur. Jednotlivé prvky pole jsou uloženy v paměti v řadě za sebou. Adresa daného prvku se dá určit pomocí jednoduchého vzorce: $adresa\_prvku = adresa\_pole + (index * velikost\_jednoho\_prvku)$. Pole lze také chápat jako specializaci asociativního pole, kde klíčem je celočíselná vzdálenost od začátku.



Obrázek 2.1: Ukázka pole.

2.3.1 Staticky alokované pole

U tohoto typu pole při vytváření musíme určit jeho velikost a ta se za běhu programu nemůže změnit. Staticky alokované pole najdeme především u programovacích jazyků se statickou kontrolou typů.

2.3.2 Dynamicky alokované pole

Dynamicky alokované pole umožní programátorovi použít pole, jehož velikost se za běhu programu může podle potřeby měnit. Tento typ pole bývá výchozím typem u skriptovacích jazyků a jazyků s dynamickou kontrolou typů. Staticky typované jazyky často mají tento typ pole jako součást základní knihovny.

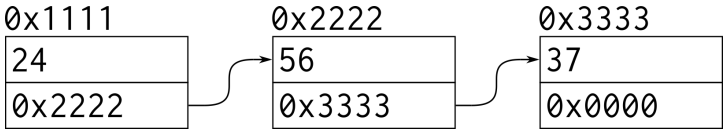
Implementace polí s dynamickou velikostí ukládají data v staticky alokovaném poli s určitou kapacitou. Operace vkládání a mazání ovlivňují hodnotu velikosti, ale pole zůstává stejné délky. Když velikost dosáhne nebo překročí kapacitu, staré hodnoty se přesunou do nového staticky alokovaného pole o větší kapacitě a staré pole se smaže.

Tabulka 2.1: Časová složitost pole. Složitost mazání odpovídá složitosti vložení.

Přístup k prvku	Vyhledání	Vložení na začátek	Vložení na konec	Vložení uprostřed
O(1)	O(n)	O(n)	O(1)	O(n)

2.4 Spojový seznam

Spojový seznam je velmi jednoduchá datová struktura, jejíž principem je, že každý prvek si uchovává odkaz na další prvek v seznamu. Seznamy mohou pojmout neomezený počet prvků a jednoduše se zvětšují, ale za cenu špatné lokality v mezipaměti procesoru. Jednotlivé prvky se mohou nacházet kdekoliv a je velice nepravděpodobné, že se všechny nacházejí v mezipaměti. V takovém případě musí procesor hodnoty načíst z podstatně pomalejší paměti RAM. [1, kapitola 2.2.3]



Obrázek 2.2: Ukázka spojového seznamu.

2.4.1 Obousměrný spojový seznam

Obousměrný seznam je varianta jednosměrného seznamu, kde každý prvek, kromě odkazu na další prvek, obsahuje ještě odkaz na prvek předchozí. Tato modifikace ulehčuje implementaci velké řady algoritmů za cenu větší spotřeby paměti. [1, kapitola 2.2.5]

Tabulka 2.2: Časová složitost spojového seznamu. Složitost mazání odpovídá složitosti vložení.

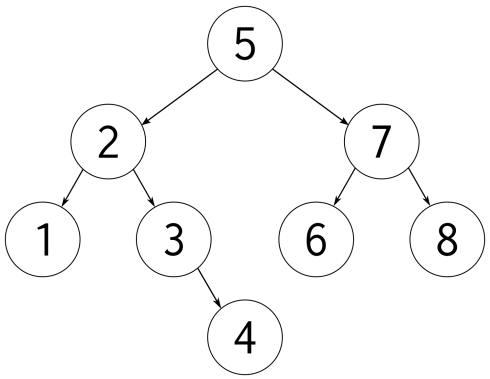
Přístup k prvku	Vyhledání	Vložení na začátek	Vložení na konec	Vložení uprostřed
O(n)	O(n)	O(1)	O(1)	O(n)

2.5 Binární strom

Binární strom je speciální variantou stromu z teorie grafů, kde každý uzel může mít maximálně dva potomky.

2.5.1 Binární vyhledávací strom

Binární vyhledávací strom je binární strom, kde hodnota každého uzlu je větší než všechny hodnoty v jeho levém podstromu a menší než hodnoty v pravém podstromu. Tato vlastnost umožňuje jednoduché a rychlé hledání hodnot, které se dají seřadit. Variace binárního vyhledávacího stromu (většinou červeno-černé stromy) jsou často použity pro implementaci asociativních polí. Použití červeno-černého stromu najdeme v C++ v `std::map` a v Javě v `TreeMap`.



Obrázek 2.3: Ukázka binárního vyhledávacího stromu.

Tabulka 2.3: Časová složitost binárního vyhledávacího stromu. Složitost mazání odpovídá složitosti vložení.

Přístup k prvku	Vyhledání	Vložení na začátek	Vložení na konec	Vložení uprostřed
O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

2.6 Hash tabulky

Hash tabulka je další implementací asociativního pole. Na rozdíl od pole, kde klíče musí být číselné hodnoty a binárního vyhledávacího stromu, kde klíče musí být seřaditelné, pro klíče v hash tabulce postačí hash funkce.

Výstupem hash funkce je (oproti potencionální velikosti vstupních dat) malé číslo, které unikátně určuje vstupní data. I když pravděpodobnost shody výsledného 64-bitového hashe u dvou různých

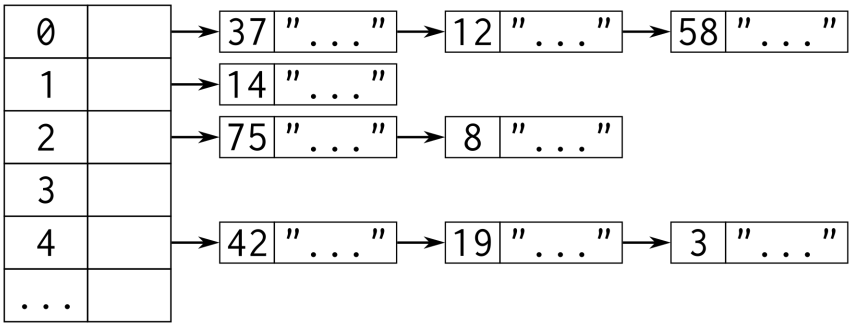
vstupů je malá, maximální velikost tabulky u většiny programů zpravidla nepřesáhne několik desítek miliónů. Z pravděpodobnosti 1 ku 2^{64} se v praxi stává naprostá jistota.

Datová struktura, u které nelze zaručit, že vložení jedné hodnoty nepřepíše jinou nesouvisející hodnotu, není moc užitečná, a proto existuje několik možných způsobů, jak tento problém vyřešit.

2.6.1 Otevřené hashování

Tabulky, které používají otevřené hashování, neukládají samotnou hodnotu, ale jinou datovou strukturu (nejčastěji spojový seznam). Tato datová struktura obsahuje prvky, které mají stejný výsledný hash. Při vyhledání hodnoty se nejdříve získá ukazatel na datovou strukturu podle zahashovaného klíče a poté se podle pravé hodnoty klíče v datové struktuře vyhledá výsledná hodnota. Na první pohled se může zdát, že lineární (nebo binární, v případě binárního stromu) vyhledávání má výrazný vliv na rychlost tabulky, ale při použití dobré hash funkce se v každé z těchto struktur bude nacházet jen velmi malé množství záznamů.

Výhodou tabulek používající tuto metodu je, že mohou pojmout více záznamů, než je jejich kapacita. Nevýhodou je větší spotřeba paměti a špatná prostorová lokalita cache.



Obrázek 2.4: Ukázka tabulky využívající otevřené hashování pomocí spojového seznamu.

2.6.2 Uzavřené hashování

Dalším schématem je uzavřené hashování, kde všechny páry klíčů a hodnot jsou uloženy přímo v interní datové struktuře tabulky. Pokud při operaci na tabulce dojde ke kolizi, výsledný hash se posunuje o předem zvolený interval, dokud se nenajde volné místo nebo správná hodnota. Interval se obvykle modifikuje podle některé z následujících možností:

- Lineární posun: jednoduchý posun o předem určené číslo (většinou 1).
- Kvadratický posun: index se postupně zvětšuje o větší a větší hodnoty (kvadraticky).
- Dvojité hashování: posun o hodnotu zvolenou jinou hash funkcí.

S vyšším zaplněním u jednoduchého lineárního posunu dochází k častému seskupování prvků, které nutí při vyhledávání projít velké množství irelevantních prvků. Kvadratický posun a dvojité

hashování sice problém seskupování z části řeší, ale jejich výkon je omezen lokalitou cache. Všechny tři varianty ale trpí při velkém zaplnění, kdy algoritmus vyhledání prodlužuje nedostatek volných míst v tabulce. Další nevýhodou je nemožnost jednoduchého vymazání prvku z tabulky. Protože prázdná místa ovlivňují průběh vyhledávacího algoritmu, vymazání může způsobit situace, kdy v tabulce je více prvků se stejným klíčem nebo v ní nelze najít existující klíč. Tento problém se řeší označením prvku jako smazaný a skutečné vymazání probíhá až při změně velikosti tabulky a následném hashování prvků.

	Klíč	
Index (pozice hashe)		Hodnota
0	14 (0)	"..."
1		
2	37 (2)	"..."
3	43 (2)	"..."
4	7 (3)	"..."

Obrázek 2.5: Ukázka tabulky využívající uzavřené hashování.

2.6.3 Kukaččí hashování (Cuckoo hashing)

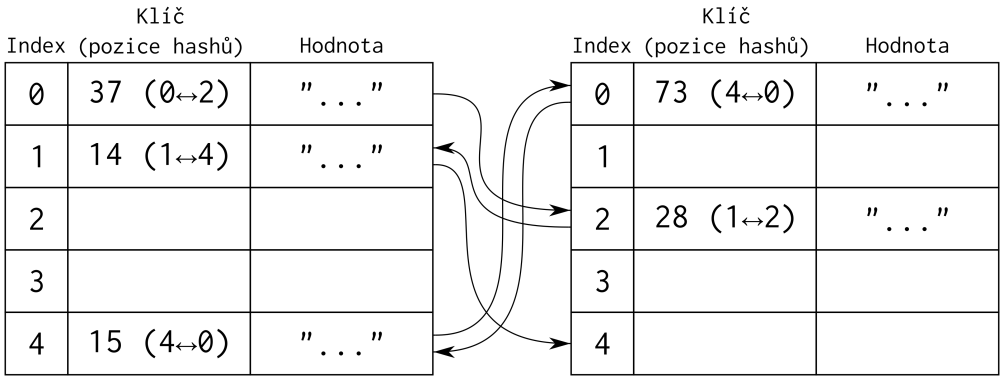
Hlavním nedostatkem uzavřeného hashování je nepředvídatelnost všech operací. Už při malém zaplnění se v tabulce mohou nacházet seskupené prvky. U většího zaplnění pak v horších případech může jednoduchý přístup k prvku prohledat velkou část tabulky. Kukaččí hashování je varianta uzavřeného hashování, která podstatně ulehčuje a zrychluje operace přístupu a mazání prvku.

Základem tohoto schématu je, že každý klíč má v tabulce více možných míst (nejčastěji dvě), kde se může nacházet. Toto je dosaženo použitím několika na sobě nezávislých a odlišných hash funkcí. Operace přístupu k prvku je v konstantním čase, protože na rozdíl od ostatních schémat nepotřebuje procházet spojovým seznamem nebo seskupenými prvky, ale pouze zkontrolovat jen předem určený počet možných míst. Stejné výhody platí i pro operaci mazání, a dokonce není potřeba prvky pouze označit jako smazané, protože žádný z prvků se nikdy nemůže nacházet na jiném místě, než které pro něj určují hash funkce.

Největší komplexita se přenáší do operace přidání prvku. Při vkládání algoritmus zkontroluje, zda alespoň jedno z možných míst pro daný hash je prázdné. V případě, že takové místo existuje, algoritmus do něj jednoduše hodnotu vloží a skončí. Pokud žádné volné místo neexistuje, algoritmus podle nespecifikovaného kritéria zvolí jedno z prohledaných míst a hodnotu na tomto místě vymění s vkládanou hodnotou. Proces vkládání se pak opakuje pro vyměněný prvek. Při špatně zvoleném kritériu, hash funkcí nebo vysokém zaplnění mohou vznikat smyčky a algoritmus nemusí nikdy

skončit. Je zapotřebí tedy smyčky detekovat nebo omezit délku běhu algoritmu. Pokud algoritmus žádné platné místo není schopen najít, probíhá zvětšení tabulky. [2, strana 3]

Schéma kukaččího hashování je možné implementovat jako jednu tabulku, ve které každý klíč má několik možných míst nebo jako několik oddělených tabulek, kde každý klíč má v každé z tabulek jedno možné místo.



Obrázek 2.6: Ukázka tabulky využívající kukaččí hashování. Šipky ukazují na další možnou pozici prvku.

2.6.4 Perfektní hashování

Hlavním příčinou kolizí v hash tabulce je samotná hash funkce. Kdybychom byli schopni vytvořit perfektní hash funkci, která nikdy nevrátí stejné hodnoty pro dva různé vstupy, dosáhli bychom přístupu k prvku v konstantním čase. Na první pohled ale už musí být jasné, že kromě komprese nikdy nebudeme schopni zakódovat $n + 1$ bitů informací do n bitů. Efektivitu hash tabulek také snižuje obvyklá implementace indexování: $index = hash \pmod{velikost\_tabulky}$, kde dva klíče, které mají rozdílný hash, sdílejí stejný index.

Obecná perfektní hash funkce pravděpodobně nikdy nebude vymyšlena, ale pro situace, kde známe všechny možné hodnoty klíčů jsme ji schopni vytvořit.

Tabulka 2.4: Časová složitost hash tabulky. Složitost mazání odpovídá složitosti vložení.

Přístup k prvku	Vyhledání	Vložení na začátek	Vložení na konec	Vložení uprostřed
O(1)	O(n)	O(1)	O(1)	O(1)

2.7 Porovnání datových struktur

Tabulka 2.5: Porovnání časových složitostí populárních datových struktur.

Datová struktura	Přístup k prvku	Vyhledání	Vložení na začátek	Vložení na konec	Vložení uprostřed
Pole	O(1)	O(n)	O(n)	O(1)	O(n)
Spojový seznam	O(n)	O(n)	O(1)	O(1)	O(n)
Binární vyhledávací strom	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Hash tabulka	O(1)	O(n)	O(1)	O(1)	O(1)

Z tabulky lze poznat, že hash tabulky jsou teoreticky stejně rychlé jako obyčejné pole. Prakticky ale proces hashování a nutnost použití otevřeného/uzavřeného hashování tabulky výrazně zpomaluje. Oproti binárnímu vyhledávacímu stromu jsou tabulky výrazně rychlejší a mají lepší lokalitu cache. Výhodou hash tabulek je také možnost použití velké řady datových typů jako klíčů. Hash funkce jsou poměrně jednoduché naprogramovat, ale ne všechny typy hodnot se dají seřadit nebo plně reprezentovat jako číslo.

Kapitola 3

Testované implementace

3.1 Standardní knihovna C++

Standardní knihovna jazyku C++ je kolekce menších knihoven, které poskytují soubor pomocných funkcí, algoritmů a datových struktur fungujících totožně, nezávisle na operačním systému a architektuře procesoru. Dnešní standardní knihovna je výsledkem mnohaletého vývoje standardovou komisí WG21. Moderní použití jazyka C++ je velmi odlišné od starého takzvaného "C s třídami". Chytré ukazatele zjednodušují práci s pamětí. Koncept "Resource Acquisition is Initialization" ulehčuje manipulaci s paměťově nebo výpočetně náročnými zdroji a standardní kontejnery ve většině případů nahrazují potřebu implementovat vlastní verze často používaných datových struktur. Dlouhá existence umožnila programátorské komunitě online vytvořit kvalitní dokumentaci a nasbírat velké množství ukázek použití, odpovědí na často kladené otázky a řešení obvyklých problémů.

Nemalá část programátorů ale nevnímá stáří standardní knihovny pozitivně. Velká řada rozhodnutí, která dávala smysl v osmdesátých a devadesátých letech minulého století, nyní aktivně škodí zájmu nových programátorů. Jazyk C++ a jeho standardní knihovna jsou dnes velkou řadou programátorů známé jen jako zastaralé a zbytečně komplikované technologie. Standardní knihovna je opravdu rozsáhlá a přesto neobsahuje například dnes naprosto nezbytné nástroje pro práci s Unicode textem.

3.1.1 std::unordered_map

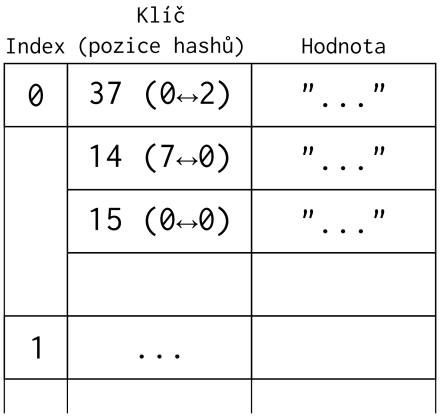
Hlavní implementací hash tabulky v jazyce C++ je `std::unordered_map`. Ačkoliv standard neurčuje jaké schéma hashování má tabulka použít, zvolené základní hodnoty a záruky pro operace silně ukazují na otevřené hashování pomocí spojového seznamu. Podobně jako většina kontejnerů standardní knihovny (kromě případů jako `std::vector<bool>`) může k datům přistupovat více vláken najednou, ale zapisovatel může být pouze jeden. [3]

3.2 Libcuckoo

Libcuckoo je knihovna poskytující výkonnou a kompaktní hash tabulku, která podporuje souběžné zápisy a čtení. [4] Hash tabulka je založena na upraveném schématu kukaččího hashování. Knihovna se jednoduše instaluje a používá, protože je distribuována jen jako kolekce hlavičkových souborů.

3.2.1 libcuckoo::cuckoohash_map

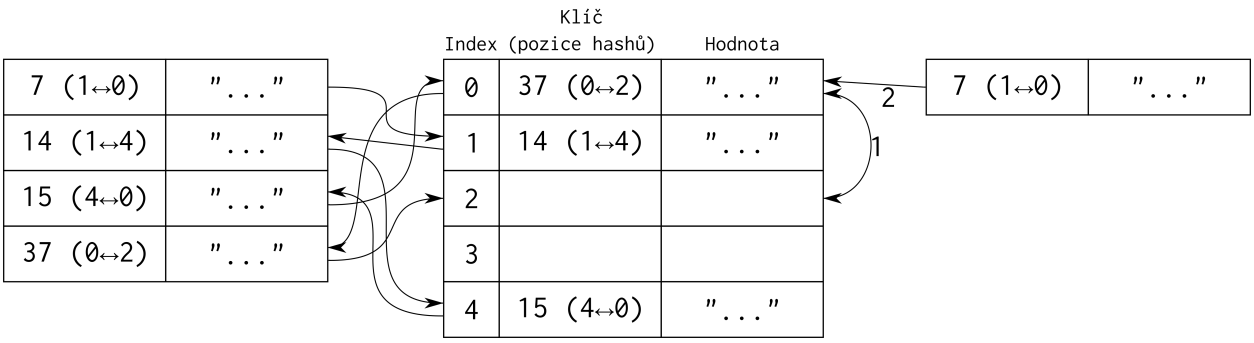
Implementace cuckoohash_map kombinuje kukaččí hashování a otevřené hashování. Slabina jednoduchého kukaččího hashování je podobná jako u jiných uzavřených schémat, velká ztráta výkonu při hustém zaplnění u operace vkládání. Cuckoohash_map používá v základu pole o velikosti 4 pro každé vnitřní místo. To znamená, že při hledání se kontroluje $2 * N(N = 4)$ možných prvků.



Obrázek 3.1: Ukázka tabulky používající interní pole. První pozice může přijmout ještě jednu položku.

Proces vkládání jednotlivých hodnot není v jednoduché implementaci kukaččího hashování vhodný pro paralelizaci. Přesouvání hodnot není atomické a souběžná operace čtení by přesouvanou hodnotu v tabulce nemusela najít. Samotný proces hledání volného místa pro přesouvané prvky je také poměrně náhodný a nezaručuje, že volné místo najde. Tento problém můžeme vyřešit tím, že nejdříve najdeme sekvenci přesunů vedoucí k prázdnému místu a teprve poté začneme prvky přesouvat. Tuto metodu, ale stále nelze použít při práci s více vlákny. Dvě vlákna vkládající různé hodnoty si mohou předem objevenou sekvenci přesunů přerušit.

Libcuckoo možnou sekvenci přesunů otáčí a místo přesouvání dvou různých prvků začíná na konci a zaměňuje vždy prvek s prázdným místem. Nemůže se tak stát, že by algoritmu zůstal náhodný prvek z tabulky, který nelze jiným vláknem najít a se kterým se musí začínat celý proces vkládání znova. Tato úprava zaručuje, že nová hodnota je vždy vložena naposledy a všechny předem existující prvky jsou dosažitelné. Pokud jiné vlákno možnou sekvenci přeruší, stačí najít sekvenci novou a pokračovat dál.



Obrázek 3.2: Ukázka vkládání prvku do tabulky. Vlevo je jednoduchá implementace náhodně vyměňující prvky. Na pravé straně se nachází Libcuckoo s nejkratší posloupností.

Dalším problémem pro kukaččí hashování je nutnost zamykat dva různé zámky. V těchto situacích si často vlákna zámky zamknou do kříže a ani jedno z nich není schopné pokračovat, nastává takzvaný deadlock. Riziko této situace je u Libcuckoo ještě větší, protože více vnitřních polí může sdílet stejný zámek. Libcuckoo tento problém řeší poměrně chytrým způsobem; všechny zámky jsou seřazené a ten s menší hodnotou se zamyká dříve.

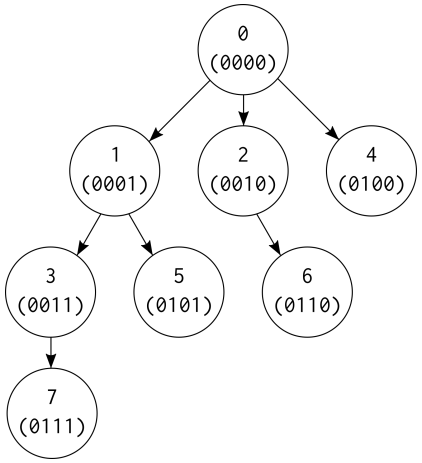
3.3 Intel oneAPI Thread Building Blocks

Intel openAPI Thread Building Blocks, dále jen oneTBB, je C++ knihovna poskytující celou řadu algoritmů, které umožňují vysokou paralelizaci. Hlavním cílem knihovny oneTBB je zjednodušit a zrychlit paralelní programování. Místo klasických vláken jsou hlavní jednotkou v oneTBB úkoly. Interní plánovač oneTBB tyto úkoly přiděluje nativním vláknům.

Kromě paralelních algoritmů poskytuje oneTBB několik paralelních datových struktur. Tyto struktury nejsou omezeny na použití pouze z oneTBB algoritmů, ale lze je použít i z nativních vláken. OneTBB poskytuje dvě různé implementace hash tabulek, `tbb::concurrent_unordered_map` a `tbb::concurrent_hash_map`.

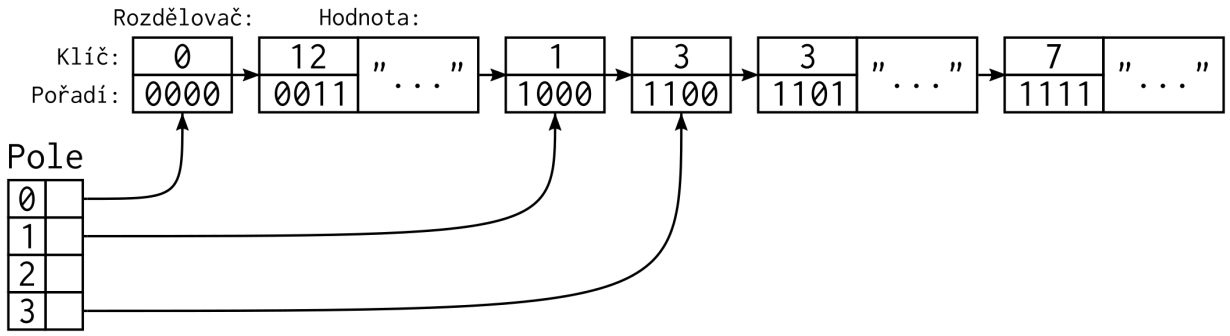
3.3.1 tbb::concurrent_unordered_map

Tato hash tabulka je založena na principu Split-Ordered Lists [5]. Hlavní datovou strukturou tabulky je spojový seznam, který obsahuje všechny vložené páry klíčů a hodnot. Prvky v seznamu jsou vzestupně seřazeny podle svého klíče, kterému je obráceno pořadí bitů a poté nastaven nově nejméně důležitý bit. Kromě hodnot se v tabulce nachází prvky, které seznam rozdělují na 2^n velikostně vyvážených částí. Tyto rozdělovače obsahují pouze číselný klíč a jsou seřazeny podle stejného principu jako normální hodnoty (pouze nemají nastavený nejméně důležitý bit). Rozdělovače v seznamu vytvářejí pomyslný strom, kde klíč rodiče každého prvku odpovídá klíči daného rozdělovače bez nejdůležitějšího nastaveného bitu (1001 -> 0001). Větší ukázka tohoto principu je ukázána v následujícím obrázku. Pokud pro klíč známe některý z rozdělovačů na správné větvi stromu (nebo obecně jakýkoliv předcházející rozdělovač), jsme schopní danou hodnotu v seznamu vyhledat.



Obrázek 3.3: Stromové uspořádání rozdělovačů.

Využití tohoto principu umožňuje druhá datová struktura. Tabulka používá pole, kde každý z prvků je odkaz na rozdělovač s klíčem rovným danému indexu prvku. Pro jakýkoliv klíč lze předcházející rozdělovač ze správné větve získat pomocí modulo hashování. Výsledný rozdělovač je získán pomocí operace klíč % velikost tabulky. Protože velikost tabulky odpovídá počtu rozdělovačů v seznamu a tato hodnota je vždy nějaký exponent dvou, operace je rovna operaci klíč & (velikost tabulky - 1). Výsledkem bude vždy nějaký platný rozdělovač, který se v seznamu nachází před daným klíčem a je ve správné větvi. Nelze ale zaručit, že to vždy bude ten nejbližší. V případě, že výsledný rozdělovač není přímým rodičem daného prvku, algoritmus musí projít větším počtem prvků, včetně hodnot ze špatných větví stromu. Efekt, který tento problém způsobuje, lze zmírnit pomocí dobře zvolené hash funkce a maximálního faktoru zaplnění. Algoritmus podporuje jen číselné klíče, ale protože nezáleží na specifickém pořadí prvků, lze ostatní klíče předem hashovat jakoukoliv hashovací funkcí (toto už `tbb::concurrent_unordered_map` dělá).



Obrázek 3.4: Ukázka tabulky založené na principu Split-Ordered Lists.

Ačkoliv princip fungování tabulky je poměrně složitý, samotná implementace je překvapivě jednoduchá. Operace vyhledávání obsahuje jednoduché modulo hashování, otočení pořadí bitů a vyhledávání v seznamu. Algoritmus začne porovnávat klíče od vypočítaného rozdělovače, dokud nenarazí na klíč větší, konec seznamu nebo na výsledný prvek. Operace přidání prvku je podobně jednoduchá. Algoritmus najde v seznamu první hodnotu s větším výsledným pořadím a před něj prvek atomicky (compare-and-swap) vloží. Operace mazání prvek v seznamu vyhledá a atomicky vymaže.

Pokud je po vložení hodnoty tabulka dostatečně naplněná, dochází k jejímu zvětšení. Algoritmus vytvoří nové pole, nakopíruje do něj staré odkazy na rozdělovače a přidá ty nové. Poté atomicky vymění ukazatel na staré pole s ukazatelem na to nové. Protože změna velikosti celou tabulku nezamyká, pokud se více vláken pokusí tabulku zvětšit, tak vlákna, které závod prohrály, musí své výsledky zahodit. Proto `tbb::concurrent_unordered_map` používá dvě optimalizace zmíněné v původním textu:

- Rozdělovače jsou rekurzivně vytvářeny až při prvním přístupu.
- Pole je rozděleno na několik menších segmentů, které jsou také vytvořeny až při prvním přístupu.

Výsledná datová struktura pak obsahuje spojový seznam a pole malých segmentů, kde každý prvek segmentu ukazuje na rozdělovač v seznamu. Protože segmenty můžeme vytvářet až při prvním přístupu k nim a jejich pořadí se nikdy nemění, změna velikosti tabulky znamená jen jednoduché zvětšení pole a vynulování ukazatelů na nové segmenty.

3.3.2 tbb::concurrent_hash_map

Na rozdíl od `tbb:concurrent_unordered_map`, kde chytrý princip umožňuje jednoduchou implementaci souběžné hash tabulky, je `tbb::concurrent_hash_map` poměrně klasickou tabulkou využívající otevřeného hashování pomocí spojového seznamu. Síla implementace spočívá v opatrném zamykání zámků. V tabulce má každý seznam a všechny jeho prvky vlastní zámek.

Kde se tato tabulka liší od ostatních implementací, je její použití přístupových objektů. Tyto přístupové objekty se starají o přístup k hodnotám, správné zamykání a odemykání zámků, a bezpečné mazání odstraněných prvků. Tabulka má dva druhy přístupových objektů:

- `tbb::concurrent_hash_map::const_accessor` - Umožňuje více vláknům souběžně číst.
- `tbb::concurrent_hash_map::accessor` - Zajišťuje exkluzivní přístup k hodnotám.

3.4 Junction

Junction je knihovna, která poskytuje čtyři implementace hash tabulek: Crude, Linear, Leapfrog a Grampa. Hlavní sílou těchto tabulek je, že řízení souběhu je založeno na lock-free technikách; tabulka tedy nepotřebuje zamykat zámky. Hlavní myšlenka, která stojí za těmito tabulkami velice zužuje typy hodnot, které se mohou použít jako klíče a hodnoty. Všechny páry klíčů a hodnot v tabulce musí být atomické. Na dnešních procesorech to znamená pouze číselné hodnoty a ukazatele. I tyto číselné hodnoty jsou ale dále omezeny. Pro každý klíč jsou navíc definované dvě hodnoty, prázdný prvek a přesměrovávací prvek (0 a 1). Další nevýhodou tabulek je špatná práce s ukazateli v hodnotách tabulky. Jako atomickou hodnotu ukazatel sice tabulka příjme, ale odstranění z tabulky samotnou hodnotu nesmaže. [6]

3.4.1 QSBR

Hlavní strukturou tabulek je pole atomických klíčů a hodnot. Ten je v implementaci uchován jako atomický ukazatel. Při zvětšení kapacity tabulky se tento ukazatel atomicky vymění za ukazatel na nové pole. Zde ale nastává problém: původní pole nemůžeme jednoduše smazat, protože ho stále mohou využívat jiná vlákna, ale nemůžeme ho v paměti ponechat, protože by to znamenalo únik paměti.

Junction volí řešení, kdy staré stavy tabulky jsou uchovány v samostatném objektu a mazány v moment, kdy žádné vlákno tabulku nepoužívá. Toho dosahuje pomocí principu quiescent state-based memory reclamation (QSBR). Podle tohoto principu jsou staré stavy tabulky rozděleny na verze. Každé vlákno využívající tabulku musí pravidelně informovat o ukončení cyklu činnosti a začátku nové verze. Teprve tehdy, až všechna vlákna pro nejstarší verzi oznámí, že ji již nepoužívají, poslední vlákno všechny staré hodnoty dané verze vyčistí. [7]

Do tohoto principu Junction zabudovává i čištění paměti uživatelských hodnot. Mimo starých hodnot tabulky může totiž programátor do aktuální verze přidat ještě funkci, která se při čištění zavolá. Po odebrání ukazatele z tabulky je možné zaregistrovat funkci, která bezpečně danou hodnotu smaže.

Nevýhodou tohoto řešení je, že uživatel musí pro správné použití těchto tabulek tuto příležitost najít nebo v horších případech dokonce vytvořit.

3.4.2 junction::ConcurrentMap_Crude

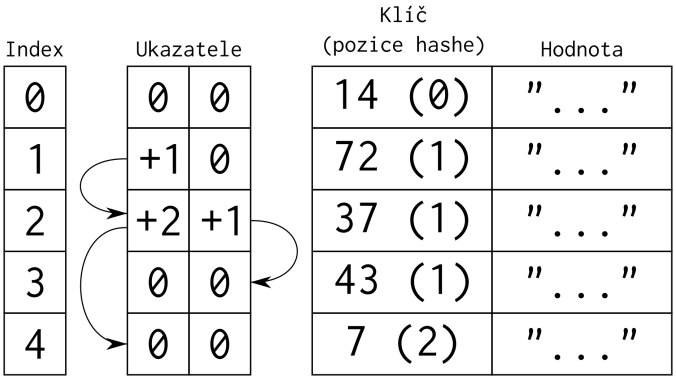
Tato tabulka je pouhou ukázkou principu stojící za implementací dalších tabulek v této knihovně. Využívá uzavřeného hashování s lineárním posunem. Nepodporuje operaci mazání specifických prvků a nelze ji zvětšit. Také nepracuje s principem QSBR.

3.4.3 junction::ConcurrentMap_Linear

Tabulka `junction::ConcurrentMap_Linear` je rozšířením `junction::ConcurrentMap_Crude`. Implementuje všechny potřebné operace (vkládání, přístup k prvku, mazání), podporuje zvětšování a QSBR.

3.4.4 junction::ConcurrentMap_Leapfrog

Tabulka využívá nového principu "Leapfrog probing", který rozšiřuje lineární posun. Každá položka ve vnitřním poli tabulky si uchovává dva číselné ukazatele na možné další prvky. První z těchto ukazatelů se odkazuje na další hodnotu v tabulce, která sdílí stejnou pozici získanou hashováním a tvoří v poli spojový seznam. Druhý ukazatel ukazuje přímo na první z prvků, jehož pozici daný prvek obsazuje. Tento druhý ukazatel tedy ukazuje na začátek předem zmíněného seznamu.



Obrázek 3.5: Ukázka "Leapfrog probing". Ukazatel na další prvek v seznamu je vpravo. Ukazatel na začátek seznamu je vlevo.

Proces přístupu k prvku pak probíhá zkontrolováním první pozice získané pomocí hashování. Pokud se hodnota zde nenachází, algoritmus zkontroluje pozici z druhého ukazatele. V tento moment se algoritmus nutně nachází ve správném seznamu. Pokud se hodnota nenachází ani na této pozici, algoritmus pokračuje s kontrolováním položek seznamu. Algoritmus končí v okamžiku, kdy jakýkoliv z ukazatelů na další prvek v seznamu je nulový.

Při vložení nového prvku se zkontroluje výsledná pozice po hashování. Pokud se na pozici už nějaký prvek nachází, algoritmus pokračuje stejnými kroky jako při přístupu k prvku. Na konci vyhledávání nebo v případě, že druhý ukazatel je nulový, algoritmus najde prázdnou pozici pomocí lineárního posunu a nastaví jeden z ukazatelů. [8]

3.4.5 junction::ConcurrentMap_Grampa

Tabulka také funguje na principu Leapfrog, ale při velkém zaplnění se rozdělí na několik menších tabulek s pevnou velikostí. I tyto tabulky se při vysokém zaplnění rozdělí na dvě další tabulky. Celá tabulka je tedy tvořena stromem tabulek.

3.5 Porovnání vlastností tabulek

Tabulka 3.1: Porovnání důležitých vlastností tabulek.

Tabulka	Souběžný přístup	Souběžný zápis	Souběžné mazání	Stabilní iterátory	Stabilní ukazatele
`std unordered_map`	Ano [1]	Ne	Ne	Ne	Ano
`libcuckoo hash_map`	Ano	Ano	Ano	Ne	Ne
`tbb unordered_map`	Ano	Ano	Ne [2]	Ano	Ano [3]
`tbb hash_map`	Ano	Ano	Ano	Ne	Ano [4]
`junction Crude`	Ano	Ano	Ne [5]	Ne [6]	Ne
`junction Linear`	Ano	Ano	Ano	Ne [7]	Ne
`junction Leapfrog`	Ano	Ano	Ano	Ne	Ne
`junction Grampa`	Ano	Ano	Ano	Ne	Ne

[1] Standardní datové struktury podporují souběžné čtení. Do tabulky ale nesmí v ten samý moment žádné vlákno zapisovat.
[2] Metoda erase je označena jako "unsafe".
[3] Není specifikováno, ale tabulka používá otevřené hashování.
[4] Není specifikováno, ale tabulka používá otevřené hashování.
[5] Tabulka nepodporuje mazání.
[6] Tabulka nepodporuje iterátory.
[7] Tabulka nepodporuje iterátory.

Kapitola 4

Praktické testování

4.1 Hardware a Software použitý při testech

Všechny testy byly provedeny na stejné konfiguraci počítače, verzi operačního systému a kompilátoru.

- Procesor: AMD Ryzen 7 3700X (8 jáder, 16 vláken)
- Paměť RAM: Kingston HyperX Fury Black 32GB (2x 16GB) DDR4 2666MHz
- Disk: Kingston Now A400 (240GB)
- Operační systém: Microsoft Windows 10 20H2 (64-bit)
- Kompilátor: Visual Studio Community 2019 (MSVC 2019)

4.2 Metodika testování

Všechny testy jsou vestavěny do jednoho programu. Jednotlivé testy se volí a konfigurují pomocí parametrů z příkazové řádky.

Každá tabulka v daném testu je zabalena do třídy, která implementuje sdílené rozhraní. Tato rozhraní jsou sdílená pouze pro tabulky v jednotlivých testech, protože mohou obsahovat pro test specifické funkce, proměnné nebo algoritmy.

Výsledky testů v grafech jsou průměry všech opakování pro jednotlivé konfigurace. S výjimkou třetího testu, který by tak zabral přes 60 hodin, jsou všechny testy opakovány 100x. Před zprůměrováním jsou odebrány dva nejlepší a dva nejhorší výsledky.

Kapitola 5

Test č. 1 - Počítání slov ve vstupní kolekci

Tento test je založen na jednoduchém počítání slov. Vstupem je soubor s textem a výstupem je tabulka, kde klíče jsou slova ze vstupní kolekce a hodnoty jsou počty jejich výskytů.

Tento problém lze jednoduše vyřešit bez použití více vláken. Nejtěžší část testu je vhodné rozdělení slov. Vstupní řetězec nelze jednoduše rozdělit pomocí mezer, protože vznikají problémy s použitím interpunkce: “ahoj”, “ahoj.”, “ahoj!” a “ahoj?” jsou podle tohoto řešení čtyři různá slova. Samozřejmě ani chytré rozdělování slov není úplným řešením. Pokud se někdo zajímá o počet výskytů slova “procesor” například v knize, tento systém by měl být schopný do celku započítat i výskyt slov “procesory” a “procesorů”. Takový systém by ale potřeboval slovník a znalost jazyka vstupního textu.

Testovací program pro tuto bakalářskou práci používá jednoduchý algoritmus, který slova rozděluje při výskytu nealfanumerických znaků. Algoritmus také ignoruje slova s nulovou délkou, aby se vyhnul případům, jako jsou slova s dvěma vykřičníky nebo otazníky.

Vstupními daty pro tento test je větší z datasetů hodnocení produktů na stránce Amazon (dostupné z https://www.kaggle.com/bittlingmayer/amazonreviews). Z datasetu jsou před spuštěním testu smazány různé Unicode symboly, které nelze převést na ASCII. Test rozděluje vláknům dataset podle řádků a očekává, že v textu nejsou žádné neplatné symboly. Ve výsledném souboru je podle utility ‘wc’ 281456447 slov na 3587717 řádcích. Test samozřejmě funguje i s neplatnými symboly, ale ty nebudou do výsledku započítány.

Výsledky algoritmu jsou jasně definované a není tedy obtížné správnost různých implementací porovnat.

5.1 Rozhraní

Potřebné metody pro tento test jsou:

- increase_or_insert - Do tabulky vloží výskyt nového slova nebo zvětší počet výskytů.
- get_key_value_pairs - Z tabulky získá pole seřazených klíčů a hodnot. Použité při kontrole výsledků.

5.2 Standardní knihovna C++

Hlavním nedostatkem `std::unordered_map` u tohoto problému je nebezpečí pádu programu při operacích souběžných s vkládáním. Pokud by test hledal v textu jen předem známá slova, tabulka by mohla být použita pomoci atomických čísel. Protože v tomto testu možná slova neznáme a tabulka neumožňuje přesnější zamykání, je nutné tabulku zamknout jako celek.

Zvolené řešení tedy tabulku zamyká pomocí mutexu při vstupu do funkce increase_or_insert a odemyká na jejím konci.

5.3 Libcuckoo a oneTBB Concurrent Hash Map

Implementovat rozhraní testu pro tabulky bylo jednoduché a proběhlo bez žádných závažných problémů.

5.4 OneTBB Concurrent Unordered Map

Na rozdíl od ostatních tabulek v tomto testu, tabulka `tbb::concurrent_unordered_map` funguje na principu lock-free a nevyužívá zamykání. K jednomu prvku tedy může najednou přistoupit dvě a více vláken. V této situaci se vyskytuje takzvaný ztracený zápis. Jednoduchá operace zvětšení hodnoty na dnes používaných architekturách není atomická a může nastat tato situace:

- Vlákno 1 přečte hodnotu x=8
- Vlákno 2 přečte hodnotu x=8
- Vlákno 1 zapíše zvětšenou hodnotu x=9
- Vlákno 2 zapíše zvětšenou hodnotu x=9

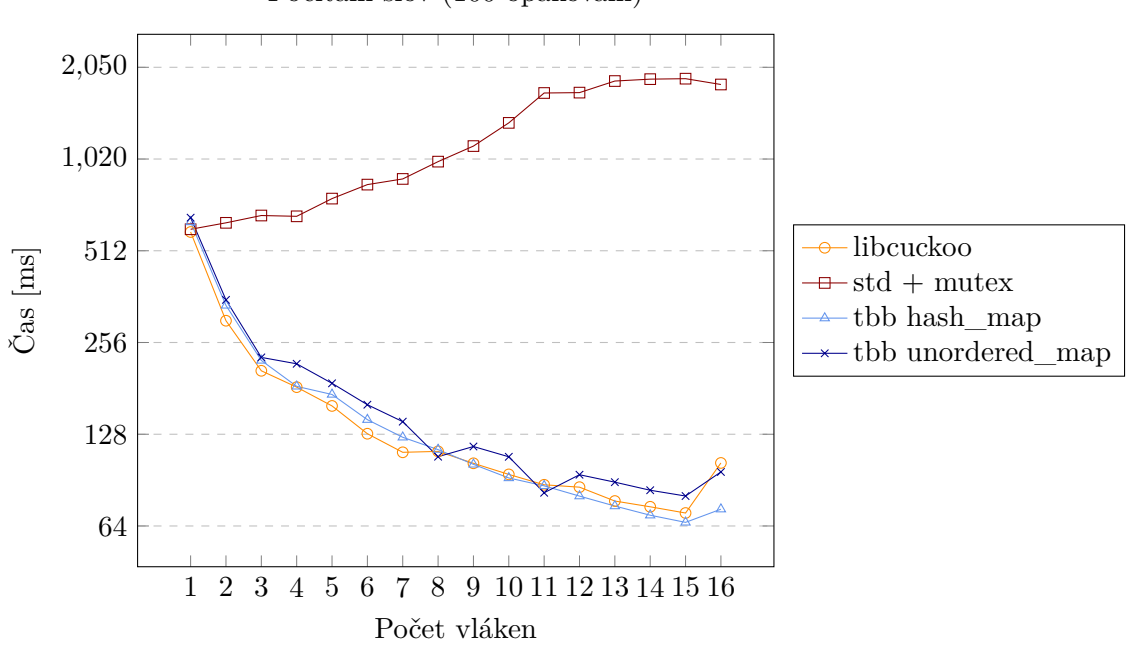
Lze vidět, že obě vlákna kvůli načasování pracují se starou hodnotou a výsledkem je místo deseti číslo devět. Z tohoto důvodu je nutné pro tabulku `tbb::concurrent_unordered_map` použít atomické číslo, který poskytuje metodu `fetch_and_increment`.

5.5 Junction

V předchozí kapitole (3.4) byl zmíněn nedostatek tabulek knihovny Junction, kde klíče a hodnoty musí být atomické. Použití řetězcových klíčů znemožňuje fungování tabulek v tomto testu.

5.6 Výsledky

Obrázek 5.1: Výsledky testu počítání slov.



U `std::unordered_map` můžeme vidět již předem očekávaný výsledek. Nutnost zamykat tabulku při každém přístupu značně zpomaluje celý test a tento efekt se zvětšuje s narůstajícím počtem vláken.

Ostatní tabulky ukazují u tohoto testu podobný výkon. Viditelně nejhorší z těchto tabulek je `tbb::concurrent_unordered_map`. Jedním z hlavních důvodu horšího výkonu jistě bude nutnost použití atomických čísel ze standardní knihovny, které mají nenulový výkonnostní efekt. Protože výsledky pro `tbb::concurrent_hash_map` a `libcuckoo::cuckoohash_map` jsou velmi podobné, je těžké u tohoto testu určit nejlepší tabulku, ale zdá se, že tabulka `tbb::concurrent_hash_map` sice začíná s horším výkonem, ale ten lépe roste s přibývajícím počtem vláken.

Kapitola 6

Test č. 2 - Hash join

Tento test implementuje algoritmus hash join, který se často využívá při vyhledávání v databázových tabulkách. Data ukládaná v těchto databázových systémech často bývají komplexní a jeden kompletní záznam muže být rozdělený mezi více tabulkami. Tyto rozdělené záznamy jsou svázány pomocí některého z atribut (nejčastěji je to nějaké identifikační číslo neboli primární/cizí klíč).

Jednoduchou ukázkou takové databáze je systém pro uchování záznamů o nákupech. Takový systém musí znát informace o zákaznících, produktech a který zákazník si koupil jaký produkt. V takovém systému musí existovat alespoň tři různé tabulky.

Tabulka 6.1: Ukázka databázové tabulky.

Zákazník	Produkt	Nákup
id_zakaz.	id_prod.	id_zakaz.
jméno	název	id_prod.
heslo	cena	datum
...	...	cena

Použití těchto tabulek zaručuje, že každý zákazník může nakoupit více produktů a každý produkt může být nakoupen více zákazníky. Pokud chceme z této databáze získat například čistý zisk všech zákazníku, je potřeba spojit tabulky "Zákazník" a "Nákup" pomocí atributu "id_zakaznika". Toto je ale $O(n * m) = O(n^2)$ operace, protože pro každého zákazníka (n) se musí zkontrolovat všechny prodeje v databázi (m).

Algoritmus "hash join" řeší právě tento rychlostní nedostatek. V první fázi ("Build") algoritmus převede jednu ze dvou spojovaných databázových tabulek (ideálně tu menší) na hash tabulku. Klíče této nové tabulky jsou atributy záznamu, podle kterých se dvě tabulky spojují a hodnoty jsou dané záznamy. Druhá fáze ("Probe") algoritmu v tabulce hledá hodnoty podle záznamů z druhé

databázové tabulky. Protože přístup k hodnotám v hash tabulkách je $O(1)$, celá časová komplexita algoritmu je $O(m+n) = O(n)$, kde m a n jsou počty záznamů ve spojovaných tabulkách.

Algoritmus ale není bez svých problémů. Velké firmy jako Google, Facebook a Amazon mají databáze, které obsahují biliony záznamů. Takové množství se často nevejde do paměti serverů a při spojování je potřeba záznamy rozdělit na menší části a použít disk.

Pro test jsou vstupní tabulky vygenerovány ze stejných dat jako pro první test. Menší z tabulek obsahuje milión primárních klíčů a náhodných řetězců ze vstupních dat. Větší tabulka obsahuje deset miliónů záznamů s primárními klíči, náhodnými řetězci a cizími klíči na záznamy z menší tabulky. Test očekává, že cizí klíč se v menší tabulce nachází.

6.1 Rozhraní

- insert - V build fázi do tabulky vloží záznam.
- get - V probe fázi z tabulky získá záznam.

6.2 Standardní knihovna C++

Protože v probe fázi nedochází k zápisu, tabulku zde není potřeba zamykat. Tabulka tedy bude pomalá při první (build) fázi, kde se zamykání nelze vyhnout, ale v druhé (probe) fázi tabulce pomůže jednodušší implementace.

6.3 Libcuckoo a tabulky knihovny oneTBB

Rozhraní tohoto testu již odpovídá existujícímu rozhraní daných tabulek.

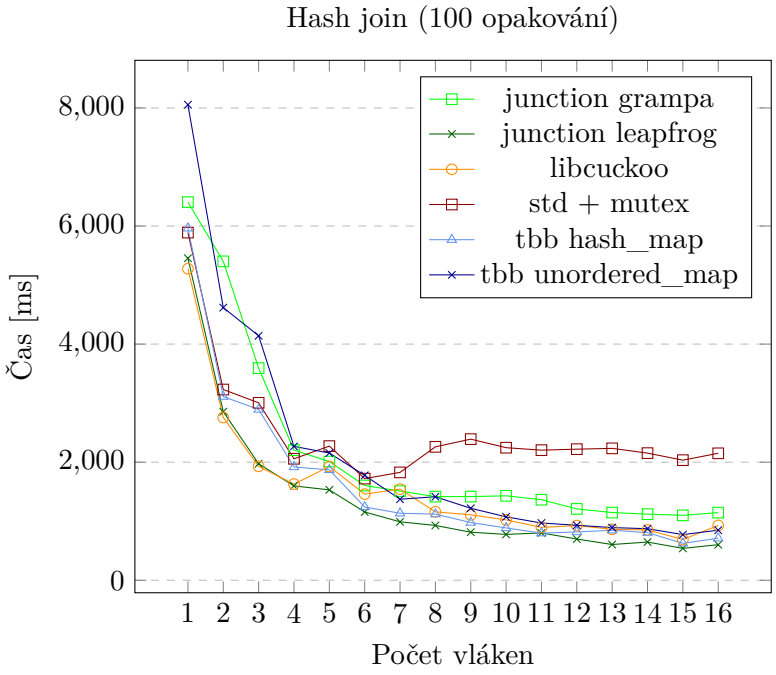
6.4 Junction

Ačkoliv u tohoto testu tabulku lze použít, knihovna zde ukazuje své nedostatky. Hodnoty tabulky nejsou atomické a musí tedy být reprezentovány jako ukazatele na objekt na haldě. Z dokumentace knihovny není jasné, zda jakákoliv z tabulek při spuštění destruktoru uzavře cyklus QSBR a spustí uložené vymazávací funkce. Obecně celý proces použití QSBR je nejasný. Podle autora knihovny se má použít v době, kdy tabulku nevyužívá žádné z vláken. U tohoto testu to znamená na konci jednoho opakování.

Z těchto důvodů jsem při implementaci testu použití QSBR u tabulek přeskočil. Znamená to tedy, že tyto tabulky mají výkonnostní výhodu, protože čištění zastaralých stavů se nezapočítává do výsledků. Alokovaná paměť je čištěna po testu pomocí iterátoru k prvkům. Tabulky `junction::ConcurrentMap_Crude` a `junction::ConcurrentMap_Linear` iterátory neposkytují a proto nejsou testovány.

6.5 Výsledky

Obrázek 6.1: Výsledky testu hash join.



Výsledky z tohoto testu potvrzují některé z poznatků z předchozího testu. Architektura tabulky `tbb::concurrent_unordered_map` ji výrazně zpomaluje oproti jiným řešením, ale ten samý princip umožňuje vysokou paralelizaci.

Tabulka knihovny Libcuckoo opět začíná na nízkém počtu vláken s dobrými výsledky, ale jejich zvyšující počet sebou nepřináší stejně velké zlepšení jako u jiných implementací.

Tabulka Leapfrog má v testu jasně nejlepší výsledky, ale na rozdíl od ostatních tabulek je čištění různých zastaralých stavů nutné řešit manuálně, a to do výsledku není započítáno. Nelze tedy určit, o kolik by se výsledky změnily, kdyby tento proces do nich byl započítán. Na druhou stranu můžeme jednoduše vidět, že stromové rozdělení tabulky Grampa výkonu škodí a tabulka je konzistentně na předposledním místě.

Překvapením je tabulka `std::unordered_map`, která se při nízkém počtu vláken drží s ostatními implementacemi. Zlomový bod přichází se 7-8mi vlákny, kde časté zamykání celé tabulky proces výrazně zpomaluje.

Největší vliv na test mají první čtyři vlákna. Poté jsou změny mezi jednotlivými testy relativně minimální.

Kapitola 7

Test č. 3 - Simulace překladové tabulky stránek v databázovém systému

Třetí test byl zamýšlen jako simulace, kde pomyslní uživatelé přistupují do systému s klíčem a ten jim vrací obsah spojený s daným klíčem. Systém měl dále udržovat maximální povolenou zaplněnost tabulky mazáním starých a dlouho nepoužívaných hodnot.

Ačkoliv test vypadá poměrně jednoduše, během implementace bylo objeveno několik zásadních problémů, kvůli kterým bylo z výsledků testů nemožné získat užitečné poznatky. Hlavním z nich je nevyváženost dvou aktérů testu, simulovaný uživatel a čistitel tabulky. Uživatelé mají v testu poměrně jednoduchou roli: opakovaně přistupují k hodnotě s náhodně vygenerovaným klíčem z předem určeného rozsahu. Role čistitele tabulky je v této simulaci časově náročná, protože k správnému smazání starých prvků je potřeba všechny prvky tabulky seřadit, například podle počtu přístupů nebo času posledního přístupu k prvku. Tento proces je komplikován tím, že většina tabulek neposkytuje stabilní iterátory, které zůstávají platné i po operacích přidání a mazání. Je tedy nutné hodnoty vyhledávat sekvenčně nebo podle náhodného klíče.

Během implementace testu jsem se problém snažil vyřešit několika způsoby. Nejnadějnější z nich bylo vytvoření seřazeného spojového seznamu, který si uchovával nejstarší prvky na začátku. Ale ani tento přístup nepomohl. Nevyrovnanost aktérů je při této variantě testu příliš velká a nebyl jsem schopen jí překonat. Výsledky testů v této podobě neukazovaly žádné patrné rozdíly mezi jednotlivými tabulkami.

Po vzájemné dohodě s vedoucím práce byl test zredukován na svou aktuální podobu. Hlavní rozdíly jsou odstranění nutnosti mazání jen starých hodnot (nyní jsou mazány sekvenčně/náhodně) a přidání čekání mezi jednotlivými přístupy do tabulky. Tato verze testu ale také neřeší zásadní problémy. Komplikovanost čistícího vlákna sice byla zmenšena, ale v testu se nenachází pouze jedno vlákno simulující uživatele. Vnucené čekání mezi přístupy do tabulky sice tento nedostatek řeší, ale v testu vytvářejí nový problém. Dvě vlákna spící 500ns provedou stejný počet operací jako jedno

vlákno, které čeká jen 250ns. Počet provedených operací tedy roste lineárně s počtem vláken do té doby, než systém není schopný vláknům včas přiřazovat práci nebo než přestane čistící vlákno stíhat.

Tyto problémy jsem nevyřešil a jejich efekt lze vidět ve výsledcích. Ve finální verzi testu je čištění řešeno pomocí dvou vláken a přistupující vlákna čekají 10000ns mezi jednotlivými přístupy k tabulce. Pokud je při vkládání nového prvku dosažena kapacita tabulky, vlákno čeká na uvolnění místa. Tabulka má v testu maximální kapacitu 500000 záznamů a lze do ní přidat klíče s maximální hodnotou o 20% větší, tedy 600000 možných klíčů.

7.1 Rozhraní

- access - Získá hodnotu z tabulky podle klíče. Pokud daná hodnota neexistuje, do tabulky ji přidá.
- erase - Z tabulky smaže hodnotu podle klíče.
- get_size - Vrátí velikost tabulky.
- get_capacity - Vrátí maximální velikost tabulky.

7.2 Standardní knihovna C++

Pro tabulku `std::unordered_map` je použit read-write zámek (`std::shared_mutex`), který podporuje několik souběžných čtenářů, ale jen jednoho zapisovatele. Protože většina času stráveného v testu by měl být přístup k prvkům, které již existují, není dobrý nápad zamykat celou tabulku. V testu se sdílený zámek získá při kontrole, zda se prvek v tabulce nachází, a exkluzivní zámek se získá pouze pokud je nutné přidat prvek do tabulky.

7.3 Libcuckoo

Tabulka je v testu použita bez žádných větších potíží.

7.4 OneTBB Concurrent Unordered Map

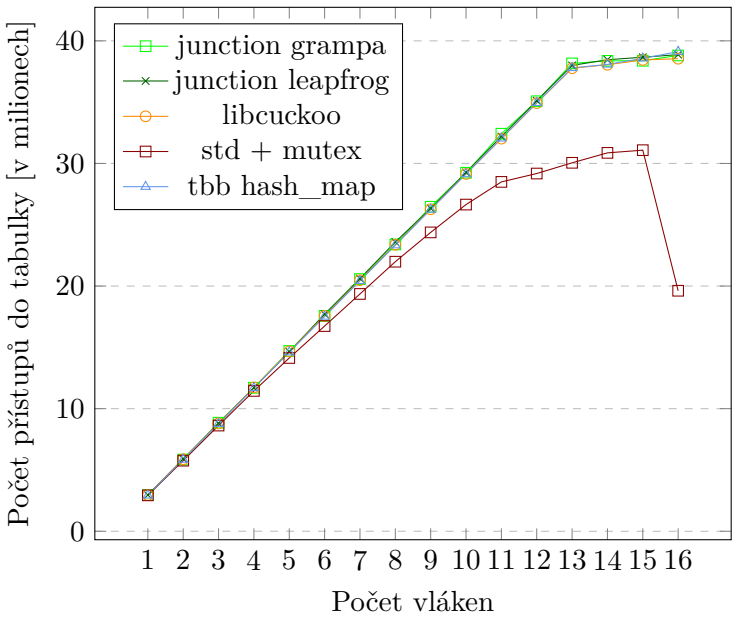
Tabulka nepodporuje bezpečné souběžné mazání prvků. Nelze ji tedy v tomto testu použít.

7.5 Junction

Pro tento test u tabulek knihovny Junction existují stejné problémy jako u předchozího testu (6.4) a ani zde se do výsledků nezapočítává možná změna výkonu, kterou by způsobilo použití QSBR.

7.6 Výsledky

Obrázek 7.1: Výsledky testu simulace překladové tabulky stránek v databázovém systému.



Ve výsledcích můžeme jednoduše vidět nevyřešené chyby v testu. S výjimkou `std::unordered_map` počet přístupů všech implementací tabulek roste lineárně s počtem vláken. Tabulku ze standardní knihovny zpomaluje nutnost zamykání sdíleného mutexu při vkládání prvků.

Kvůli nepřesnosti plánovačů procesů je v testu využit spinlock zámek. Procesor aktivně dokola kontroluje, zda může cyklus opustit a pokračovat dále v předem běžící části programu. Tento typ zámku ale zabírá procesorový čas, který mohl operační systém využit jinak. V případě tohoto testu to znamená, že vlákno, které do tabulky vkládá prvek, čeká, než se v tabulce místo uvolní. Těchto vláken může takto čekat více a zabírat čas, který mohlo využít jedno z čistících vláken.

Tento efekt můžeme vidět v menším měřítku u všech tabulek od třinácti vláken. Ve větším měřítku se projevuje jako hluboký propad výkonu u šestnácti vláken u tabulky `std::unordered_map`. Operační systém nezvládá přiřazovat dvěma čistícím vláknům dostatek času oproti šestnácti vláknům, které k hodnotám v tabulce pouze přistupují. Ve staré verzi testu, kde v programu bylo jen jedno čistící vlákno, byl propad výrazně větší, protože operační systém měl menší šanci probudit správné vlákno.

Kapitola 8

Závěr

Cílem této bakalářské práce bylo porovnat výkon několika implementací hash tabulek a myslím, že s výjimkou posledního testu se z výsledků dají získat užitečné poznatky. Datové struktury standardní knihovny nejsou používané pro vysoký výkon, ale protože poskytují konzistentní funkcionalitu a předvídatelný výkon napříč širokou škálou platforem a operačních systémů.

Knihovna Libcuckoo poskytuje tabulku s výborným výkonem a jednoduchým rozhraním. I přes nedostatek kvalitní dokumentace se tabulka jednoduše používá a neobsahuje žádné neočekávané nedostatky nebo nepopsané vlastnosti. Instalace je jednoduchá a bezproblémová.

Na rozdíl od ostatních knihoven dostává Intel oneTBB pravidelné aktualizace a profesionální podporu. Tabulka `tbb::concurrent_hash_map` při větším počtu vláken dosahuje ještě lepšího výkonu než Libcuckoo a výsledky `tbb::concurrent_unordered_map` jsou sice horší, ale pořád srovnatelné s ostatními implementacemi. Nevýhodou ale je, že tabulky jsou distribuovány jen jako součástí celé oneTBB knihovny, která je poměrně obsáhlá a obsahuje velké množství dalších datových struktur a algoritmů, které daný projekt nemusí nutně využít.

Myslím, že knihovny Libcuckoo a oneTBB nabízí skvělé hash tabulky, které mají podobné profily výkonu a záleží jen na preferenci každého programátor, kterou z tabulek si zvolí. To samé ale nemůžu říct o knihovně Junction. Nemožnost použití řetězcových klíčů razantně omezuje počet případů, ve kterých se tabulky dají využít. Princip QSBR je sice zajímavý, ale jeho správné použití není tak jednoduché, jak na svém blogu autor popisuje. Tabulky navíc obsahují rozdíly v rozhraní a nelze jednoduše nahradit jednu tabulku za druhou (tabulka crude nepodporuje iterátory; parametr začáteční velikosti tabulky Leapfrog musí být exponent dvou, jinak program padá; tabulka Grampa parametr počáteční velikosti ignoruje). Autor na svém blogu [6] tabulku popisuje jako experimentální a já s ním musím souhlasit. Junction tabulky sice fungují na zajímavém principu, ale nemůžu doporučit jejich použití ve skutečných programech.

Literatura

1. KNUTH, Donald E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0201896834.
2. FAN, Bin; ANDERSEN, David G.; KAMINSKY, Michael. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)* [online]. 2013 [cit. 2021-04-29]. Dostupné z: http://www.cs.cmu.edu/~dga/papers/memc3-nsdi2013.pdf.
3. ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* [online]. Fifth. 2017 [cit. 2021-04-29]. Dostupné z: https://www.iso.org/standard/68564.html.
4. GOYAL, Manu; FAN, Bin; LI, Xiaozhou; ANDERSEN, David G.; KAMINSKY, Michael. *Libcuckoo* [online]. Carnegie Mellon University, Intel Corporation, and GitHub, 2018 [cit. 2021-04-29]. Dostupné z: https://github.com/efficient/libcuckoo.
5. SHALEV, Ori; SHAVIT, Nir. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM* [online]. 2006-06, vol. 53, no. 3, s. 379–405 [cit. 2021-04-29]. ISSN 0004-5411. Dostupné z DOI: 10.1145/1147954.1147958.
6. PRESHING, Jeff. *New Concurrent Hash Maps for C++* [online]. 2016-02 [cit. 2021-04-29]. Dostupné z: https://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/.
7. PRESHING, Jeff. *Using Quiescent States to Reclaim Memory* [online]. 2016-03 [cit. 2021-04-29]. Dostupné z: https://preshing.com/20160726/using-quiescent-states-to-reclaim-memory/.
8. PRESHING, Jeff. *Leapfrog probing* [online]. 2016-03 [cit. 2021-04-29]. Dostupné z: https://preshing.com/20160314/leapfrog-probing/.

Příloha A

Zdrojový kód testovacího programu

K bakalářské práci je přiložen ZIP soubor, který obsahuje zdrojový kód a pomocné skripty k testům, které byly implementovány pro tuto práci. Vstupní data pro testy nejsou součástí archívu. Data se dají stáhnout z https://www.kaggle.com/bittlingmayer/amazonreviews, nebo je možné použít vlastní data (za předpokladu, že následují formát specifikovaný v páté kapitole).

Po extrahování archívu a zkompilování kódu by měla výsledná adresářová struktura vypadat přibližně takto:

- bin - Adresář s výsledným programem.
  - HashmapBenchmark(.exe) - Testovací program.
  - run_benchmarks.py - Předem nakonfigurovaný skript, který spustí všechny testy.
- data - Adresář se vstupními daty pro testy.
  - process_amazon.py - Skript, který upraví data pro první test.
  - create_hash_join_data.py - Skript, který vytvoří vstupní data pro druhý test.
- src - Adresář se zdrojovým kódem pro testovací program.